UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

DIM0538 - ARQUITETURAS REATIVAS

Relatório de Desenvolvimento de microsserviços com WebFlux, Event Drive e Padrões de Resiliência.

Matrícula: 20200047120

Nome: Michelle Teixeira Martins

1. **Introdução**

A implementação de uma arquitetura voltada a eventos (Event Drive) tem sido cada vez mais necessária nos sistemas atuais, visto que o alto processamento de dados faz com que cada vez mais as aplicações busquem uma alternativa viável para a implementação de serviços escaláveis. Em conjunto a isso, existe a necessidade de garantir que tais sistemas sejam também resilientes a falhas.

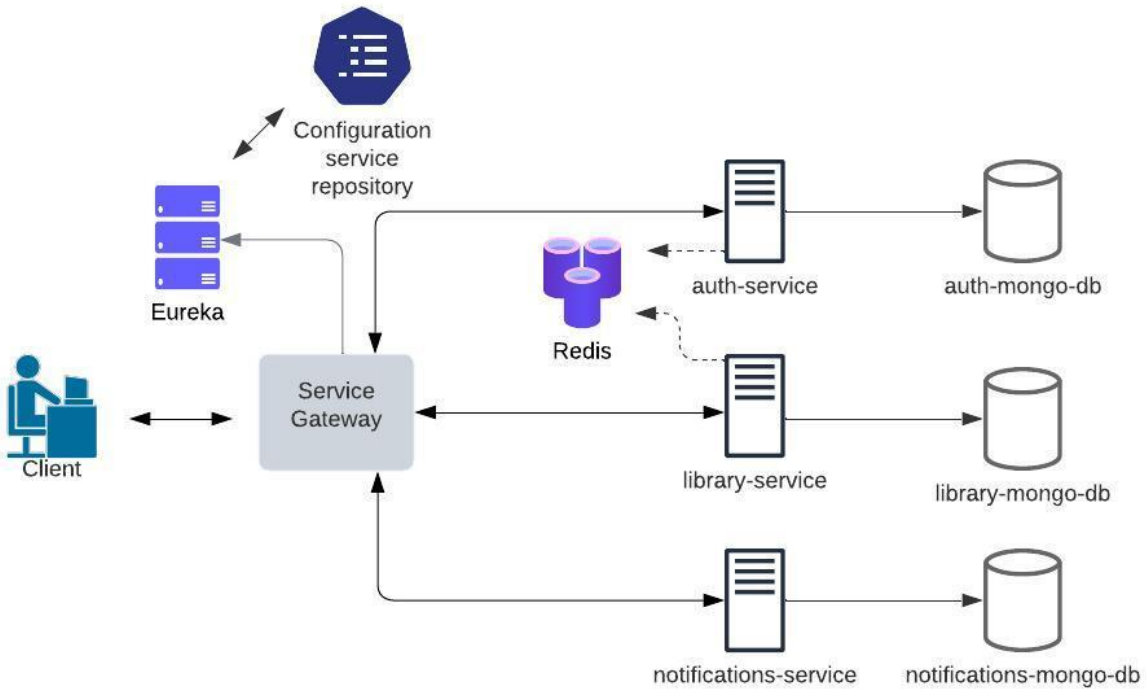
Nesse viés a adoção de uma arquitetura de event drive em conjunto com padrões de resiliência proporcionam a um sistema de microsserviços inúmeras vantagens, sendo estas, a aquisição de responsividade, tolerância a falhas, facilidade de manutenção e melhora na manutenção dos dados. Sendo assim, o seguinte relatório busca realizar a análise do impacto de tais ferramentas em uma aplicação construída com paradgmas de programação reativa.

Principais tecnologias usadas:

- Spring Reactive Web
- Spring Data Reactive MongoDB
- MongoDB
- Java Mail Sender (envio de email)
- WebClient (comunicação cliente servidor)
- Eureka Discovery Client
- Cloud LoadBalancer
- Spring Cloud Stream
- Spring Cloud Function
- Resilience4j

Arquitetura:

Foram elaborados seis microsserviços, sendo três deles serviços de API (Interface de Programação de Aplicação) que são um serviço para livraria virtual, um serviço de autenticação de usuários e um serviço para envio de email e notificação. Os demais serviços atuam como auxiliares, sendo eles um serviço que disponibiliza um repositório de configuração, um serviço para registro e descoberta e um serviço de gateway. Ademais existem a integração com banco de dados mongoDB e redis.



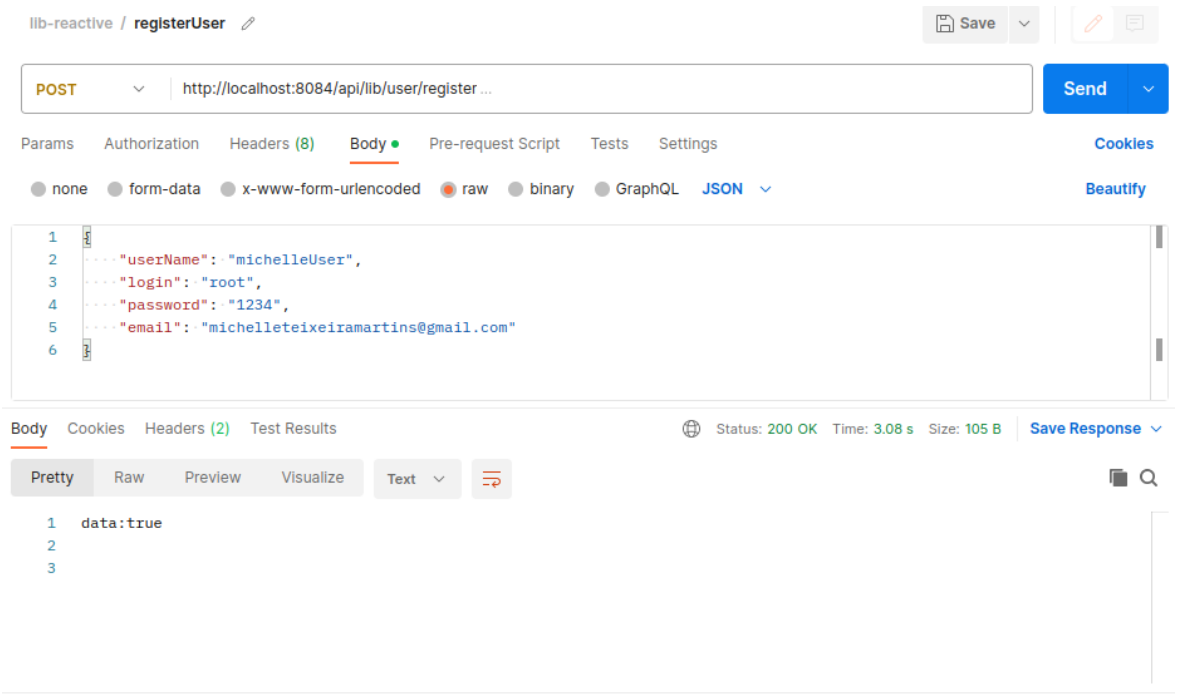
2. **Aplicação de Event Drive**

   Com intuito de simplificar a aplicação de uma arquitetura orientada a eventos, utilizou-se o framework Spring Cloud Stream em conjunto com o Spring Cloud Function.

   2.1 Integração com RabbitMQ

   Para controlar o fluxo de mensagens optou-se por realizar o suporte do Spring Cloud Stream na integração com o broker de mensageria RabbitMQ, de modo que o seguinte fluxo foi aplicado:

   - **Novo cliente é cadastrado no sistema de biblioteca:**



- **Publicador gera um novo evento para o serviço de notificação:**

```
1 usage   new *
public Mono<Boolean> processMessage(String emailFrom, String emailTo, String title, String text){
    Message<UserService.SendEmail> emailEvent = MessageBuilder
            .withPayload(new UserService.SendEmail(emailFrom, emailTo, title, text))
            .setHeader(MessageHeaders.CONTENT_TYPE, MimeTypeUtils.APPLICATION_JSON).build();
    return Mono.just(streamBridge.send(message_event, emailEvent));
}
```

- **Consumidor presente no serviço de notificação recebe esse evento e gera a notificação que será enviada para o cliente:**

3. **Aplicando padrões de resiliência**

Com intuito de simplificar a adesão de uma arquitetura resiliente, utilizou-se os recursos para padrões de resiliência disponibilizados pela biblioteca Java Resilience4j.

3.1 uso de Circuits Breakers e Retry

- Anotações para uso de Circuit Breaker e Retry:

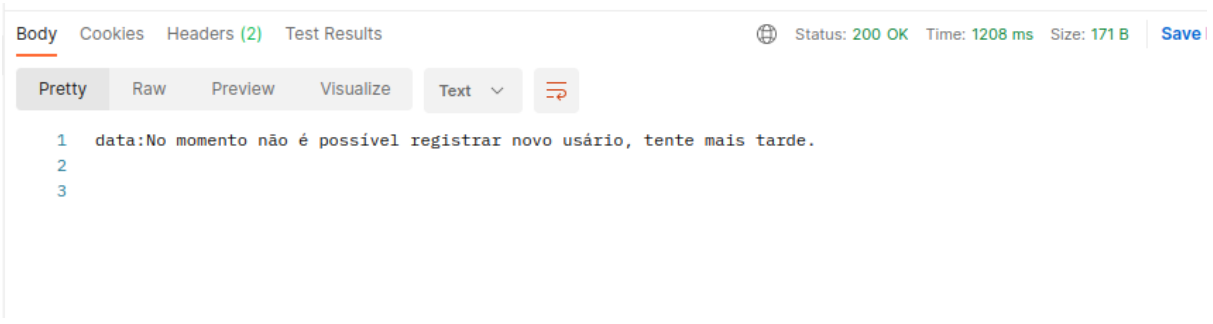
```
michelle
@GetMapping(value="users", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
@Retry(name = "retfryService", fallbackMethod = "fallbackCache")
public Flux<String> getAllUsers() { return userService.getAllUsers(); }

michelle *
@CircuitBreaker(name= "circuitBreakerService", fallbackMethod = "fallbackRegister")
@Retry(name = "retryService", fallbackMethod = "fallbackRegister")
@PostMapping(value="register", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Mono<Boolean> register(@RequestBody UserService.UserRegister user){
    return userService.registryUser(user)
            .flatMap(e-> {
                try {
                    Gson gson = new Gson();
                    String jsonString = e.replaceAll( regex: "data:",  replacement: "");
                    UserModel object = gson.fromJson(jsonString, UserModel.class);
                    return userService.save(object)
                            .then(userService.processNewUserMessage(user.email()));
                } catch (Exception ex) {
                    throw new RuntimeException(ex);
                }
            }).doOnNext(e-> System.out.println("resultado" + e));
}
```

- Visualização dos parâmetros do circuit breaker pelo actuator:


```
"circuitBreakers": {
    "status": "UP",
    "details": {
        "circuitBreakerService": {
            "status": "UP",
            "details": {
                "failureRate": "-1.0%",
                "failureRateThreshold": "30.0%",
                "slowCallRate": "-1.0%",
                "slowCallRateThreshold": "100.0%",
                "bufferedCalls": 3,
                "slowCalls": 0,
                "slowFailedCalls": 0,
                "failedCalls": 2,
                "notPermittedCalls": 0,
                "state": "CLOSED"
            }
        }
```

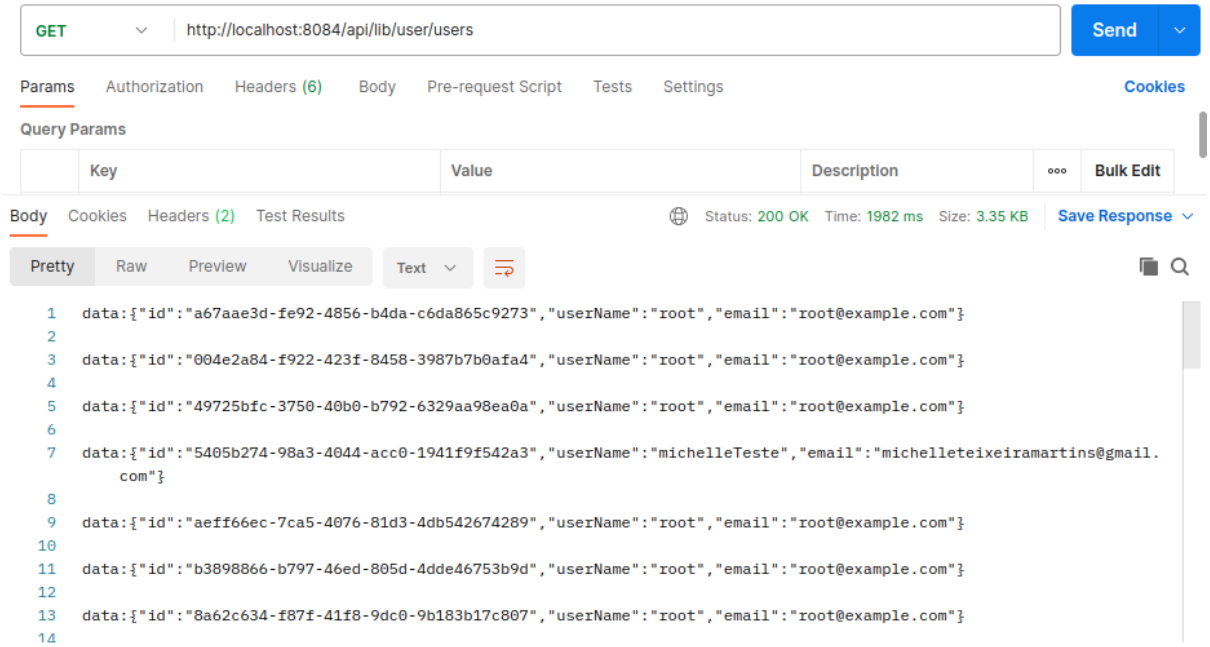

3.2 Tratamento de erro personalizado com uso de fallbacks

- Retornar uma mensagem de erro indicando que o serviço de autenticação não está disponível:



- Carregar os dados salvos em cache e retornar no corpo da requisição:

```
no usages   new *
public Flux<UserModel> fallbackCache(Throwable error){
    System.out.println("falback: carregando usuários do cache;");
    return userCache.getAll();
}
```



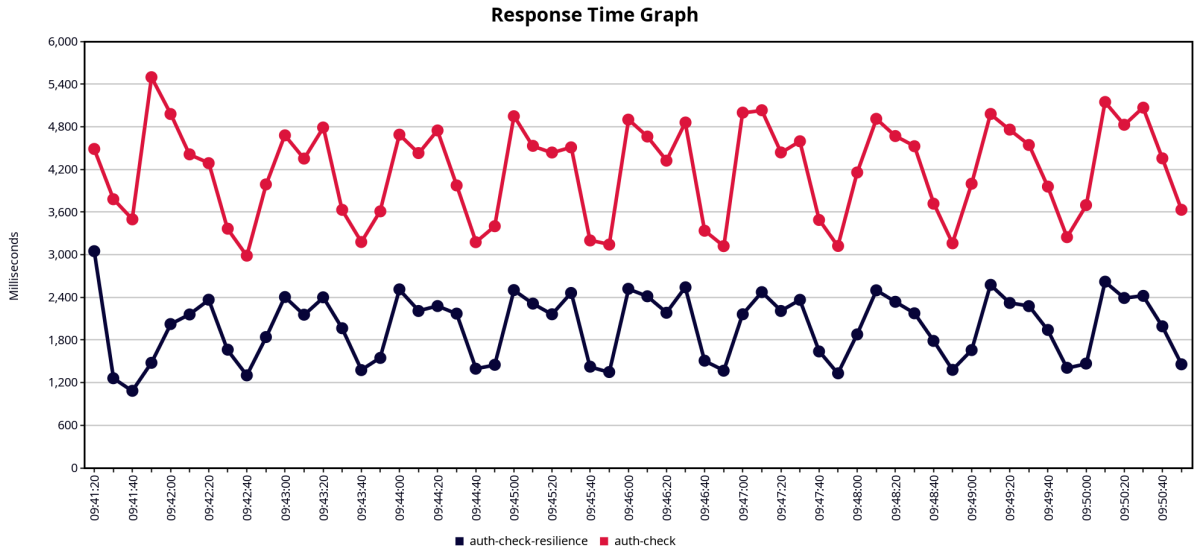
4. Resultados e experimentos

Para a realização dos experimentos foi utilizado inicialmente uma máquina com as seguintes especificações:

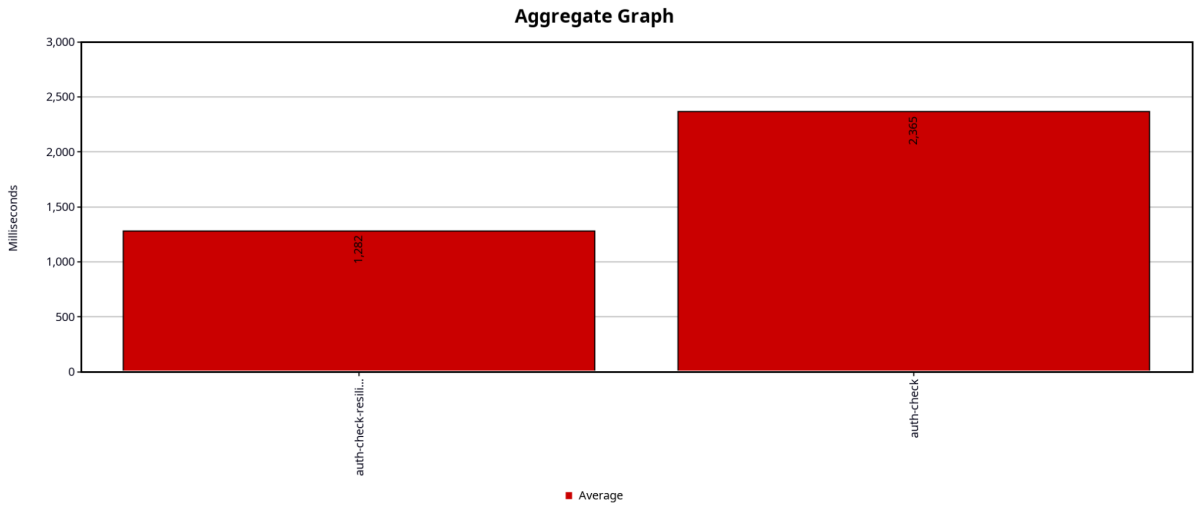
Computador	Especificações
Computador	Samsung
Processador	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Memória	8GB
Sistema Operacional	**Especificações**
Sistema	Manjaro Linux
Versão	6.1.23-1-MANJARO (64-bit)
Kernel	Linux
Arquitetura	x86-64

Os experimentos foram realizados de modo a aplicar testes de carga (via jmeter) na aplicação. Sendo assim, foram usados dois endpoints, um com aplicação de padrões de resiliência (auth-check-resilience) e outro sem o uso de métodos para controle de resiliência (auth-check).

- **Tempo de resposta**

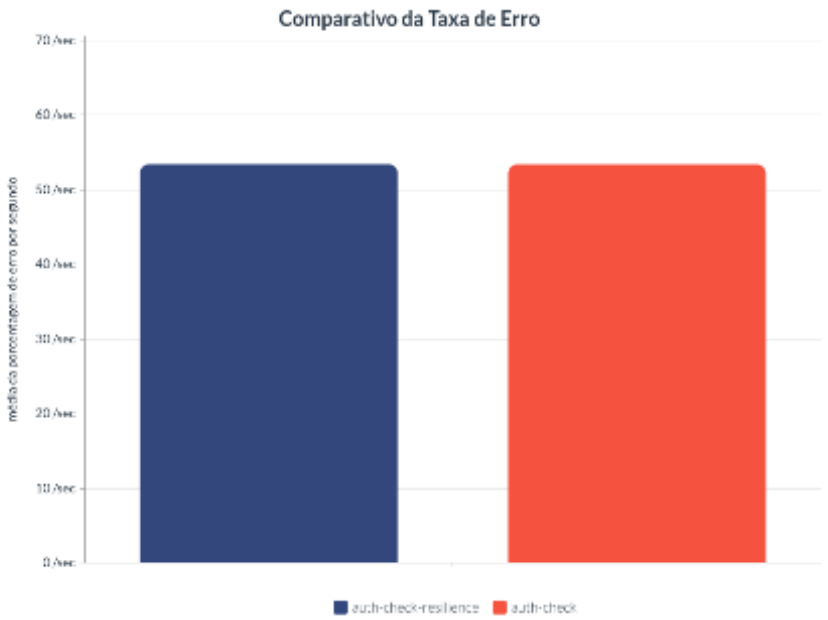


Comparação de tempo de resposta



Comparação de média do tempo de resposta

- **Percentual de erro**



Comparação da média na taxa de erro

5. Conclusão

Tendo em vista os resultados nos testes de carga, podemos concluir que apesar de apresentarem taxas de erro relativas o endpoint com suporte a padrões de resiliência mostrou um desempenho melhor no tempo de resposta se comparado com o endpoint sem padrões de resiliência. Ademais, nota-se que a adesão de event driven trouxe benefícios, visto que facilitou a geração de funcionalidades no sistema e reduziu o acoplamento entre os serviços.

Sendo assim, podemos concluir que aderir uma arquitetura voltada a eventos e com suporte a padrões de resiliência são fatores importantes e que devem ser levados em consideração no processo de criação e gerenciamento de aplicações reativas.

6. Referências

- **Link para repositório com API's reativas:** https://github.com/Michelletxr/reactive-applications
- **Link para repositório com serviço de configuração, gateway e eureka:** https://github.com/Michelletxr/reactive-applications/tree/master/reactive-applications/aux_services